

# CS1010S Programming Methodology

## Lecture 7

# Searching & Sorting

10 Oct 2018

# Mid-term Survey



# Saying of the Wise

*I hear and I forget.*

*I see and I remember.*

*I do and I understand.*

- *Xun Zi*

不聞不若聞之  
見之不若見之  
知之不若知之  
學至于行之而止矣



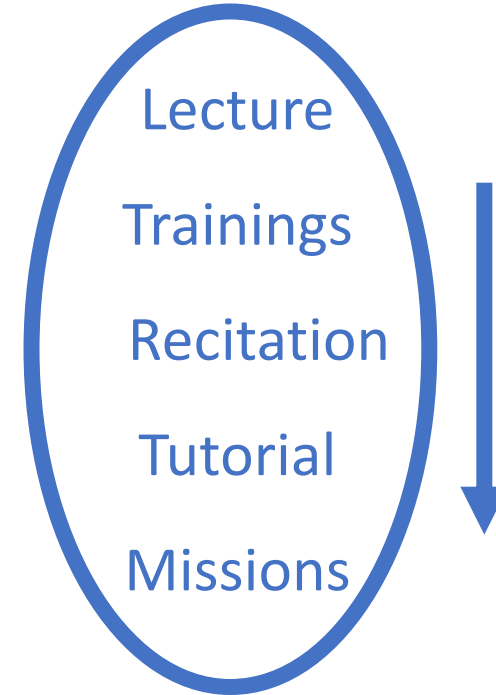
# Some Philosophy

Python is like chess

- Easy to learn, hard to master

Levels of learning

- Knowledge
- Understanding
- Application



Secret Recipe for Success in CS1010S

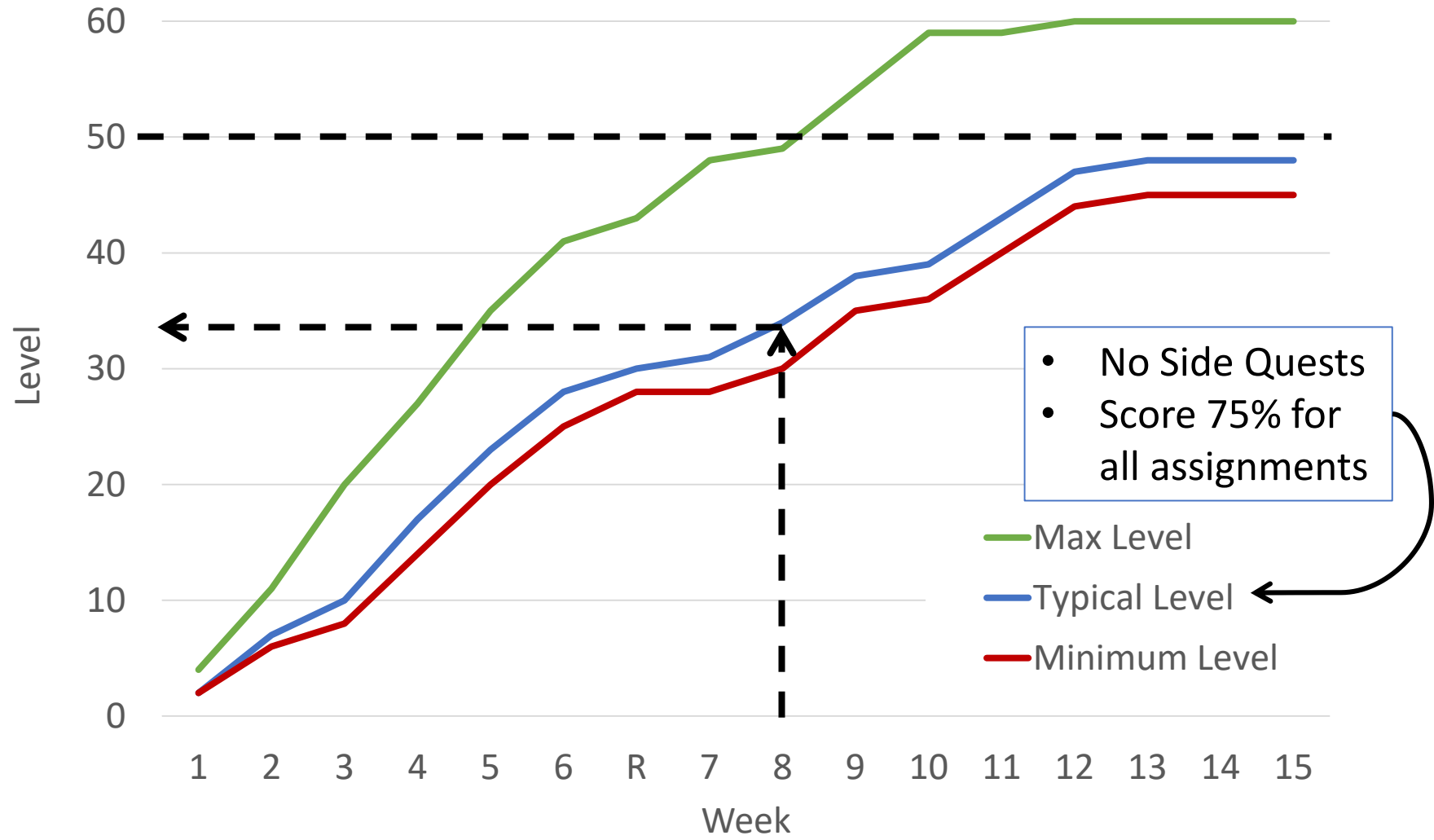
- DOING IT (Getting hands dirty and writing code)!

Lots of Optional  
Trainings

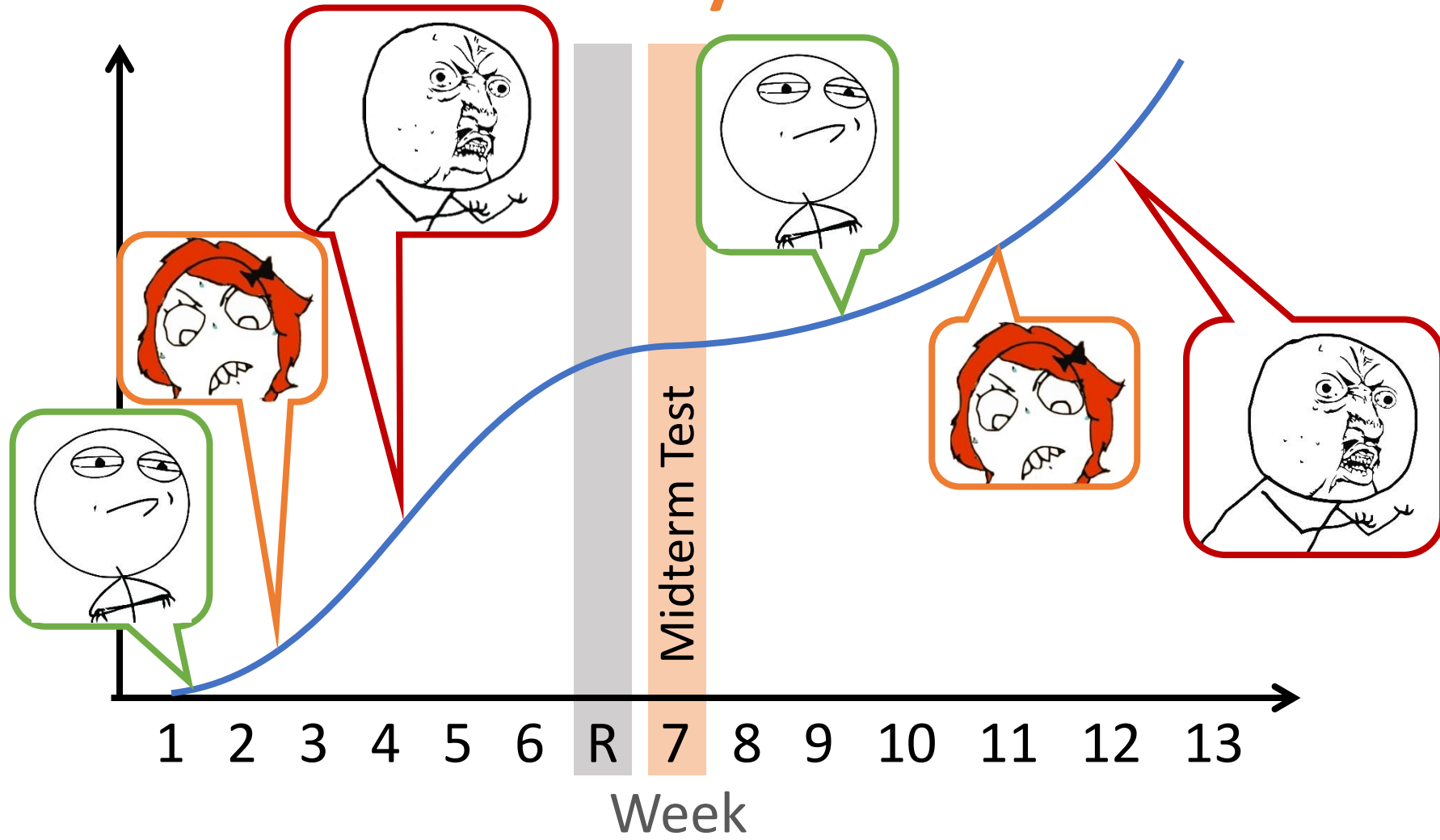
# Remedial

## Check Announcements

# Expected Level Progression



# Difficulty Curve





# Plagiarism

The act of presenting another's work or idea that as your own.



# Today's Agenda

- Python Lists
- Searching
  - Linear search
  - Binary search
- Sorting
  - Basic sorting algorithms
  - Properties of sorting

# Why Lists?

- Recall: tuples are **immutable**.

```
int_tuple = (1, 2, 3)
```

```
int_tuple[0] = 5
```

**TypeError: 'tuple' object does not support item assignment**

- What about lists?

```
int_list = [1, 2, 3] # this is a list
```

```
int_list[0] = 5      # [5, 2, 3]
```

**Mutable!!**

a.k.a. Arrays

Lists are sequences  
So are tuples and strings

# Sequence Operations

## Tuple

```
()  
(1, 2, 3)  
type((1, 2, 3))  
<class 'tuple'>
```

```
t = tuple(range(5))  
→ (0, 1, 2, 3, 4)  
t[4] → 4  
t[2:] → (2, 3, 4)
```

## List

```
[]  
[1, 2, 3]  
type([1, 2, 3])  
<class 'list'>
```

```
l = list(range(5))  
→ [0, 1, 2, 3, 4]  
l[4] → 4  
l[2:] → [2, 3, 4]
```

# Sequence Operations

## Tuple

a = (1, 2, 3, 4)

b = (5, 6, 7, 8)

c = a + b

c → (1, 2, 3, 4, 5, 6, 7, 8)

## List

a = [1, 2, 3, 4]

b = [5, 6, 7, 8]

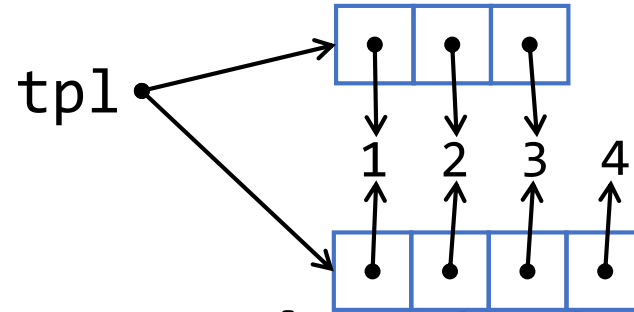
c = a + b

c → [1, 2, 3, 4, 5, 6, 7, 8]

# Appending element to list

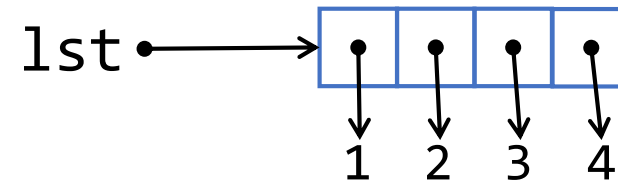
With Tuples, we can only create new tuples

```
tp1 = (1, 2, 3)
tp1 = tp1 + (4,)
```



With Lists, we can directly **append** to the list

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
```



↖  
element

No reassignment necessary

# Extending list with list

We can also directly **extend** an existing list

```
lst = [1, 2, 3]      list
lst.extend([4, 5, 6])
lst                  → [1, 2, 3, 4, 5, 6]
```

This is **equivalent** to the following

```
lst = [1, 2, 3]
lst += [4, 5, 6]
lst                  → [1, 2, 3, 4, 5, 6]
```



# Mutable versus Immutable

```
lst = [1, 2, 3]
```

```
lst2 = lst
```

```
lst == lst2
```

→ True

```
lst is lst2
```

→ True

```
lst += [4, 5, 6]
```

```
lst
```

→ [1, 2, 3, 4, 5, 6]

```
lst2
```

→ [1, 2, 3, 4, 5, 6]

```
lst == lst2
```

→ True

```
lst is lst2
```

→ True

# Mutable versus Immutable

```
tup = (1, 2, 3)
```

```
tup2 = tup
```

```
tup == tup2
```

→ True

```
tup is tup2
```

→ True

```
tup += (4, 5, 6)
```

```
tup
```

(1, 2, 3, 4, 5, 6)

```
tup2
```

(1, 2, 3)

```
tup == tup2
```

→ False

```
tup is tup2
```

→ False

# Deleting Elements

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
```

```
>>> del a[0]
```

```
>>> a
```

```
[1, 66.25, 333, 333, 1234.5]
```

```
>>> del a[2:4]
```

```
>>> a
```

```
[1, 66.25, 1234.5]
```

```
>>> del a[:]
```

```
>>> a
```

```
[]
```

What if we do `del a`?

# Sequence Operations

Some other handy functions that will work on sequences

|                                 |                                   |
|---------------------------------|-----------------------------------|
| <code>len([1, 2, 3, 4])</code>  | <code>→ 4</code>                  |
| <code>min([1, 2, 3, 4])</code>  | <code>→ 1</code>                  |
| <code>max([1, 2, 3, 4])</code>  | <code>→ 4</code>                  |
| <code>1 in [1, 2, 3]</code>     | <code>→ True</code>               |
| <code>5 not in [1, 2, 3]</code> | <code>→ True</code>               |
| <code>[1, 2, 3] * 2</code>      | <code>→ [1, 2, 3, 1, 2, 3]</code> |

# List Operations

```
lst = [1, 2, 3, 4]
```

```
lst.copy()
```

returns a shallow copy of a list

```
lst.insert(<pos>, <element>)
```

inserts **element** into position **pos**

# List Operations

`lst.pop(<pos>)`

returns and remove element at position `pos`. If `pos` is omitted, removes the last element

`lst.remove(<element>)`

removes first occurrence of `element` from list, error if `element` is not in the list.

`lst.clear()`

clears the list

# List Operations

```
s = [1, 2, 3, 4, 5]
t = s.copy()      # t = [1, 2, 3, 4, 5]
s.reverse()       # s = [5, 4, 3, 2, 1]
                  # t = [1, 2, 3, 4, 5]
s.insert(0, 1)     # s = [1, 5, 4, 3, 2, 1]
s.pop()            # 1
                  # s = [1, 5, 4, 3, 2]
s.pop(1)           # 5
                  # s = [1, 4, 3, 2]
s.insert(2, 2)     # s = [1, 4, 2, 3, 2]
s.remove(4)        # s = [1, 2, 3, 2]
s.clear()          # s = []
```

# Iterating on Lists

We can iterate through lists using `for` operator

```
>>> s = [1, 2, 3, 4, 5]
>>> for element in s:
    print(element)
```

1

2

3

4

5



# List Comprehension

We can iterate also generate new lists

```
>>> s = [1, 2, 3, 4, 5]
>>> t = [n**2 for n in s]
[1, 4, 9, 16, 25]
```

Does this look familiar?

It should. This is equivalent to:

```
t = list(map(lambda n: n**2, s))
```

# List Comprehension

Together with a filter

```
>>> s = [1, 2, 3, 4, 5]
>>> t = [n**2 for n in s if n%2]
[1, 9, 25]
```

This is equivalent to:

```
t = list(map(lambda n: n**2,
             filter(lambda n: n%2), s))
```

# Python Lists: Summary

- Lists are **sequences**
  - can be used with all the sequence operations
- Lists are **mutable**
  - has mutable operations which are not common to tuples and strings

# Searching

# Searching

- You have a list.
- How do you find something in the list?
- **Basic idea:** go through the list from start to finish one element at a time.

# Linear Search

- Idea: go through the list from start to finish

|   |   |   |   |
|---|---|---|---|
| 5 | 2 | 3 | 4 |
|---|---|---|---|

- Example: Search for 3

|   |   |   |   |
|---|---|---|---|
| 5 | 2 | 3 | 4 |
|---|---|---|---|

3 not found, move on

|   |   |   |   |
|---|---|---|---|
| 5 | 2 | 3 | 4 |
|---|---|---|---|

3 not found, move on

|   |   |   |   |
|---|---|---|---|
| 5 | 2 | 3 | 4 |
|---|---|---|---|

Found 3.

# Linear Search

Idea: go through the list from start to finish

```
# equivalent code
for i in [5, 2, 3, 4]:
    if i == 3:
        return True
```

# Linear Search

Implemented as a function:

```
def linear_search(value, lst):  
    for i in lst:  
        if i == value:  
            return True  
    return False
```

What kind of performance can we expect?

Large vs small lists?

Sorted vs unsorted lists?

$O(n)$



Can we do better?

Of course Ia!

# Searching

IDEA:

If the elements in the list were sorted in order, life would be much easier.

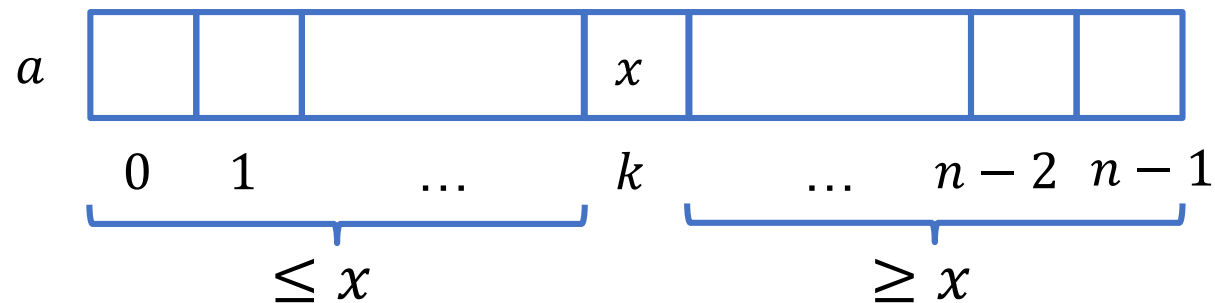
Why?

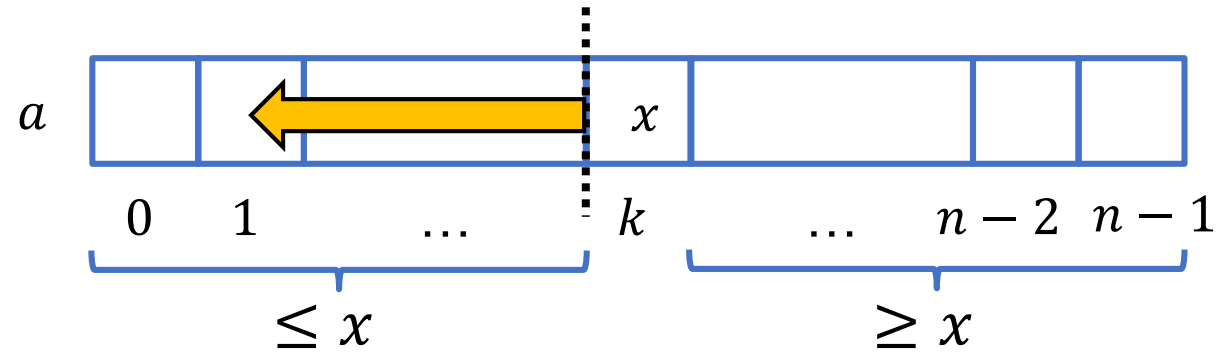


# IDEA

If list is sorted, we can  
“divide-and-conquer”

Assuming a list is sorted in ascending order:





if the  $k^{\text{th}}$  element is larger than what we are looking for, then we only need to search in the indices  $< k$

# Binary Search

1. Find the middle element.
2. If it is what we are looking for (key), return **True**.
3. If our key is smaller than the middle element, repeat search on the left of the list.
4. Else, repeat search on the right of the list.

# Binary Search

Looking for 25 (key)

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

Find the middle element: 34

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

Not the thing we're looking for:  $34 \neq 25$

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

$25 < 34$ , so we repeat our search on the left half:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

# Binary Search

Find the middle element: 12

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

$25 > 12$ , so we repeat the search on the right half:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

Find the middle element: 25

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

Great success: 25 is what we want

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|



# Binary Search

“Divide and Conquer”

In large sorted lists, performs much better than linear search on average.

# Binary Search

Algorithm (assume sorted list):

1. Find the middle element.
2. If it is we are looking for (key), return True.
3. A) If our key is **smaller** than the middle element, repeat search on the left of the element.  
B) Else, repeat search on the right of the element.

# Binary Search

```
def binary_search(key, seq):  
    if seq == []:  
        return False  
    mid = len(seq) // 2  
    if key == seq[mid]:  
        return True  
    elif key < seq[mid]:  
        return binary_search(key, seq[:mid])  
    else:  
        return binary_search(key, seq[mid+1:])
```

# Binary Search

```
def binary_search(key, seq): # seq is sorted
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2 # get middle
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high):  
        if low > high:  
            return False  
        mid = (low + high) // 2  
        if key == seq[mid]:  
            return True  
        elif key < seq[mid]:  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

Step 1. Find the  
middle element.

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high):  
        if low > high:  
            return False  
        mid = (low + high) // 2  
        if key == seq[mid]:  
            return True  
        elif key < seq[mid]:  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

key → 11

helper(0, 10-1)

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high): # 0, 9  
        if low > high:  
            return False  
        mid = (low + high) // 2 # mid=4  
        if key == seq[mid]:  
            return True  
        elif key < seq[mid]:  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

Step 1. Find the  
middle element.

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):
    def helper(low, high): # 0, 9
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]: # 11 == 25
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 2. If it is  
what we are  
looking for,  
return True



# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):
    def helper(low, high): # 0, 9
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]: # 11 == 25
            return True
        elif key < seq[mid]: # 11 < 25
            return helper(low, mid-1) # helper(0, 4-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is  
smaller, look at  
left side

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=4
        if key == seq[mid]: # 11 == 25
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key is  
smaller, look at  
left side

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high): # 0, 3  
        if low > high:  
            return False  
        mid = (low + high) // 2 # mid=1  
        if key == seq[mid]:  
            return True  
        elif key < seq[mid]:  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

Step 1. Find the  
middle element

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):
    def helper(low, high): # 0, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=1
        if key == seq[mid]: # 11 == 9
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 2. If it is  
what we are  
looking for,  
return True

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high): # 0, 3  
        if low > high:  
            return False  
        mid = (low + high) // 2 # mid=1  
        if key == seq[mid]: # 11 == 9  
            return True  
        elif key < seq[mid]: # 11 < 9  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

Step 3a. If key  
is smaller, look  
at left side

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high): # 0, 3  
        if low > high:  
            return False  
        mid = (low + high) // 2 # mid=1  
        if key == seq[mid]: # 11 == 9  
            return True  
        elif key < seq[mid]: # 11 < 9  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high) # helper(1+1, 3)  
    return helper(0, len(seq)-1)
```

Step 3b. Else  
look at right  
side

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high): # 2, 3  
        if low > high:  
            return False  
        mid = (low + high) // 2  
        if key == seq[mid]:  
            return True  
        elif key < seq[mid]:  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

Step 3b. Else  
look at right  
side

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high): # 2, 3  
        if low > high:  
            return False  
        mid = (low + high) // 2  
        if key == seq[mid]:  
            return True  
        elif key < seq[mid]:  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

Step 3b. Else  
look at right  
side



# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high): # 2, 3  
        if low > high:  
            return False  
        mid = (low + high) // 2 # mid=2  
        if key == seq[mid]:  
            return True  
        elif key < seq[mid]:  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

Step 1. Find  
the middle  
element

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=2
        if key == seq[mid]: # 11 == 12
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 2. If it is  
what we are  
looking for,  
return True

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):
    def helper(low, high): # 2, 3
        if low > high:
            return False
        mid = (low + high) // 2 # mid=2
        if key == seq[mid]: # 11 == 12
            return True
        elif key < seq[mid]: # 11 < 12
            return helper(low, mid-1) # helper(2, 2-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

Step 3a. If key  
is smaller, look  
at left side

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high): # 2, 1  
        if low > high:  
            return False  
        mid = (low + high) // 2  
        if key == seq[mid]:  
            return True  
        elif key < seq[mid]:  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

Step 3a. If key  
is smaller, look  
at left side

# Binary Search

Now let's try searching for 11:

|   |   |    |    |    |    |    |     |     |     |
|---|---|----|----|----|----|----|-----|-----|-----|
| 5 | 9 | 12 | 18 | 25 | 34 | 85 | 100 | 123 | 345 |
|---|---|----|----|----|----|----|-----|-----|-----|

```
def binary_search(key, seq):  
    def helper(low, high): # 2, 1  
        if low > high: # 2 > 1  
            return False  
        mid = (low + high) // 2  
        if key == seq[mid]:  
            return True  
        elif key < seq[mid]:  
            return helper(low, mid-1)  
        else:  
            return helper(mid+1, high)  
    return helper(0, len(seq)-1)
```

Key cannot be  
found. Return  
False

# Binary Search

- Each step eliminates the problem size by half.
  - The problem size gets reduced to 1 very quickly
- This is a simple yet powerful strategy, of halving the solution space in each step
- What is the order of growth?

$$O(\log n)$$

# Wishful Thinking

We assumed the list was sorted.

Now, let's deal with this assumption!

# Sorting



# Sorting

- High-level idea:
  1. some objects
  2. function that can order two objects

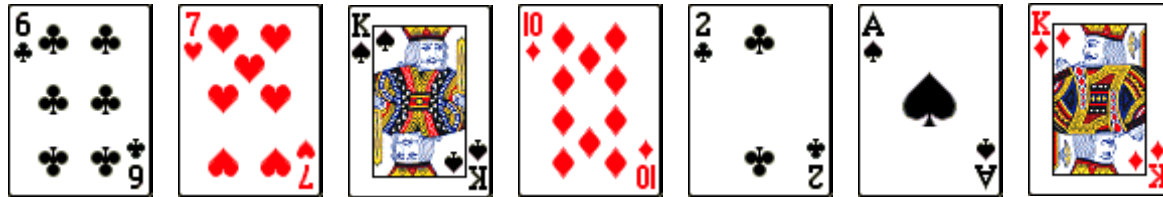
⇒ order all the objects

How Many Ways to  
Sort?

Too many. 😊

# Example

Let's sort some playing cards?

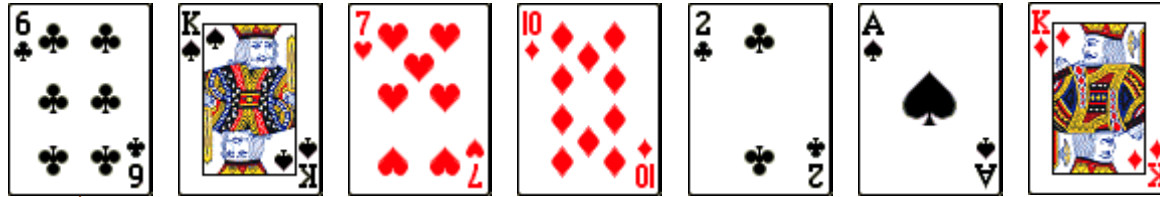


What do you do when  
you play cards?

# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

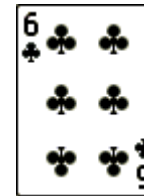
Unsorted



Sorted



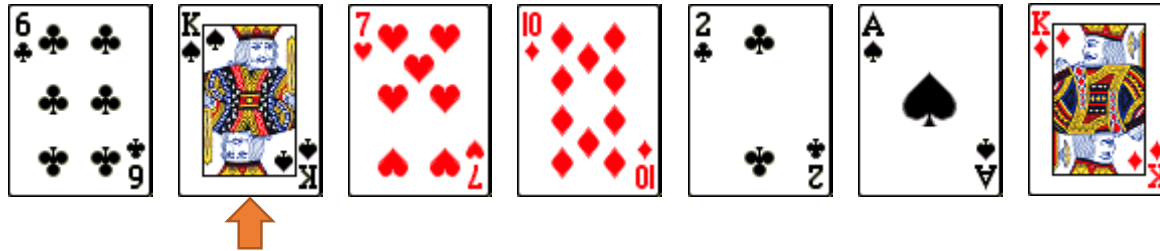
Smallest



# Obvious Way

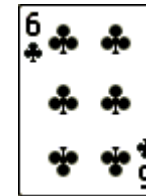
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

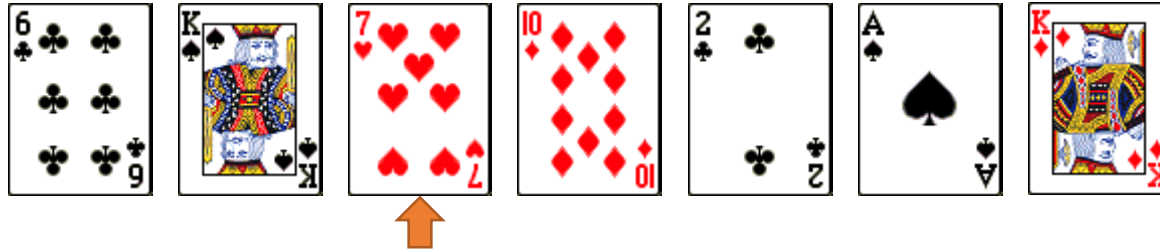
Smallest



# Obvious Way

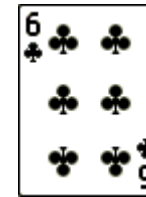
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

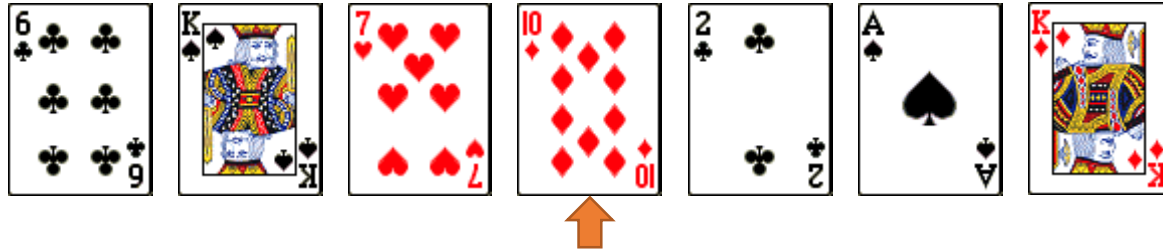
Smallest



# Obvious Way

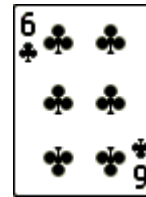
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

Smallest

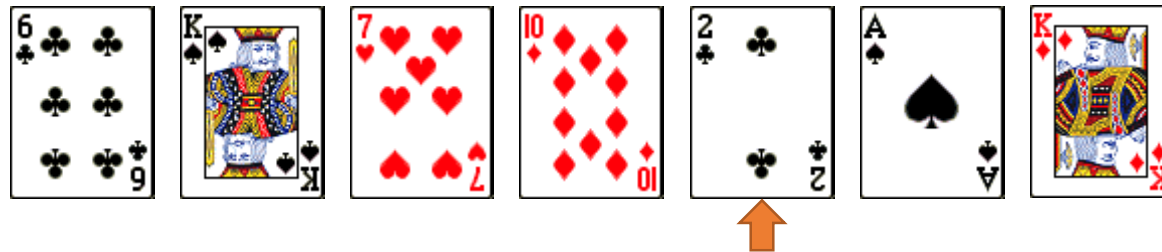




# Obvious Way

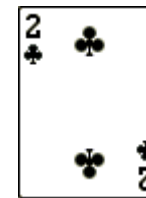
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

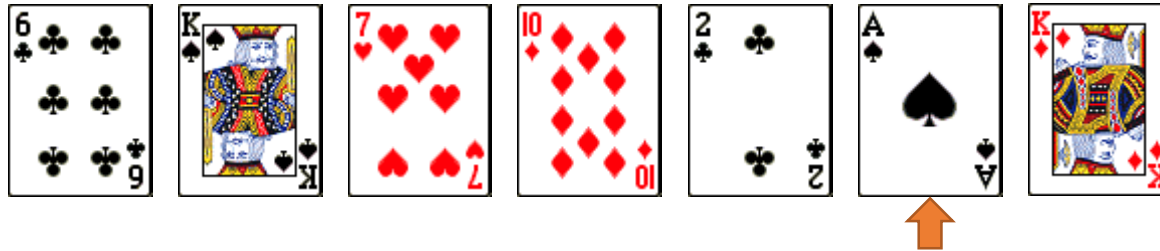
Smallest



# Obvious Way

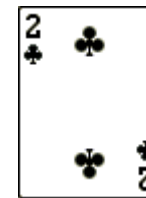
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

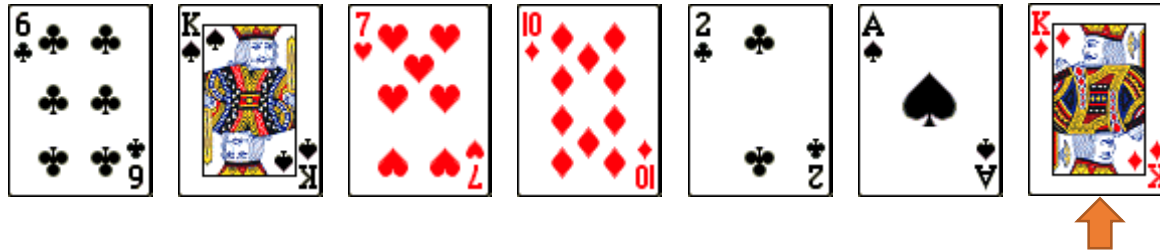
Smallest



# Obvious Way

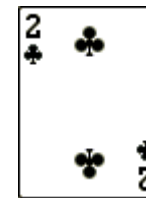
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

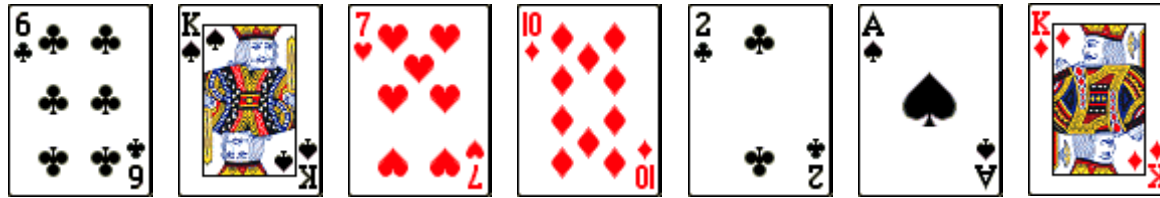
Smallest



# Obvious Way

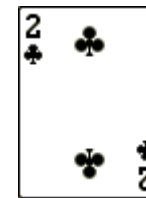
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

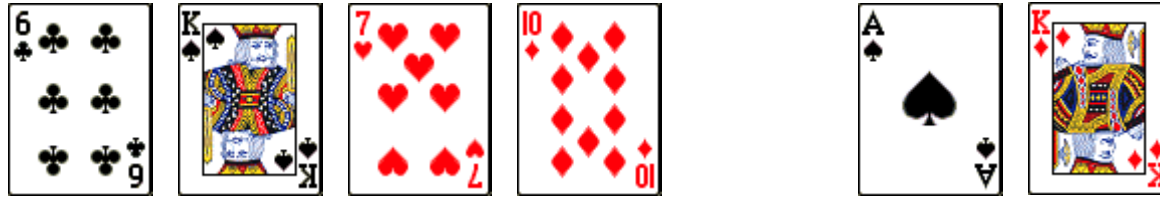
Smallest



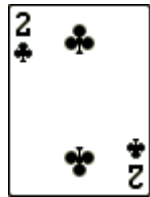
# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted



Sorted

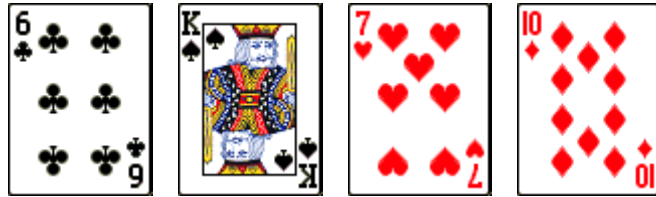


Smallest

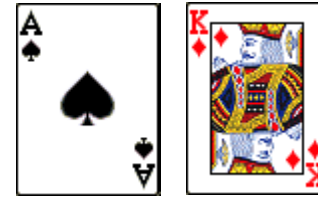
# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

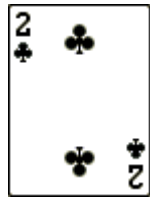
Unsorted



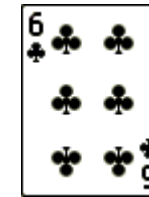
Repeat



Sorted



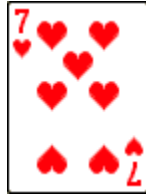
Smallest



# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

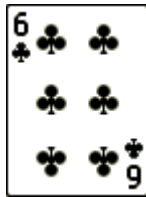
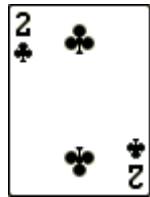
Unsorted



Repeat



Sorted

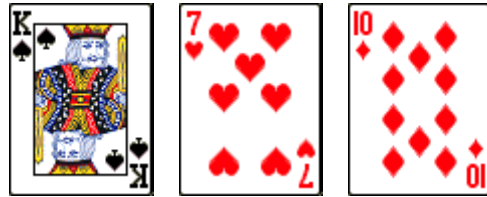


Smallest

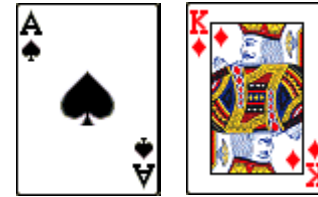
# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

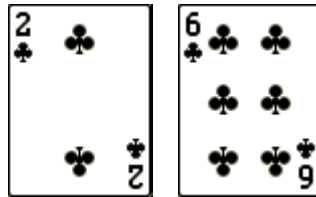
Unsorted



Repeat



Sorted



Smallest





# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

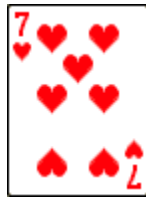
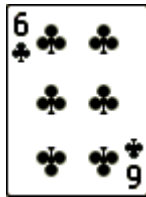
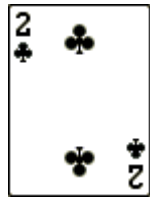
Unsorted



Repeat



Sorted



Smallest

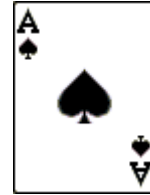
# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

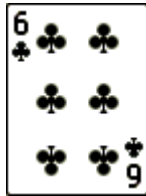
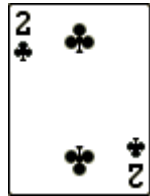
Unsorted



Repeat



Sorted



Smallest



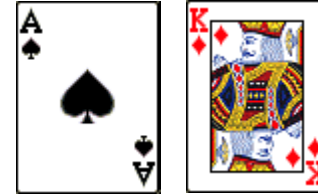
# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

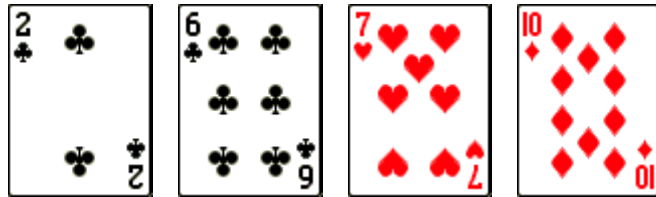
Unsorted



Repeat



Sorted



Smallest

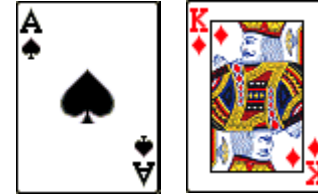
# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

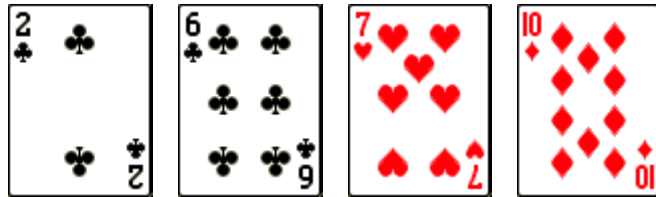
Unsorted



Repeat



Sorted



Smallest



# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

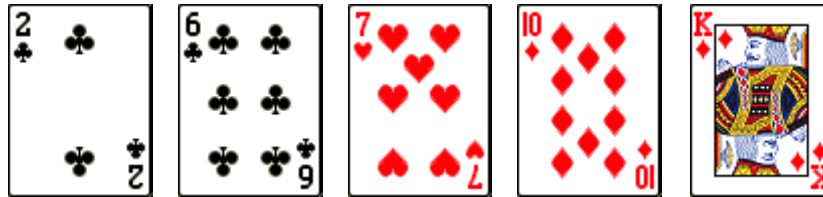
Unsorted



Repeat



Sorted



Smallest

# Obvious Way

Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

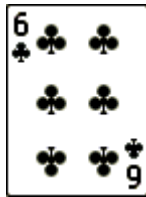
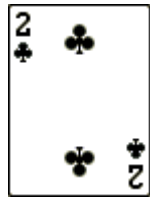
Unsorted



Repeat



Sorted



Smallest



# Obvious Way

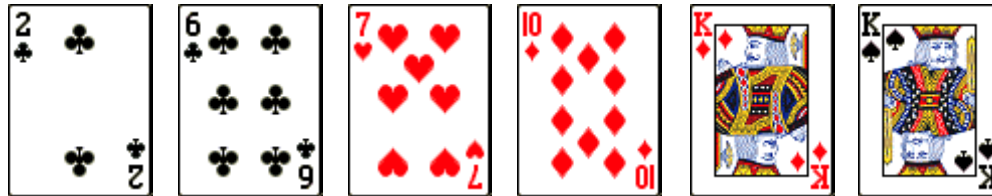
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted

Repeat



Sorted



Smallest

# Obvious Way

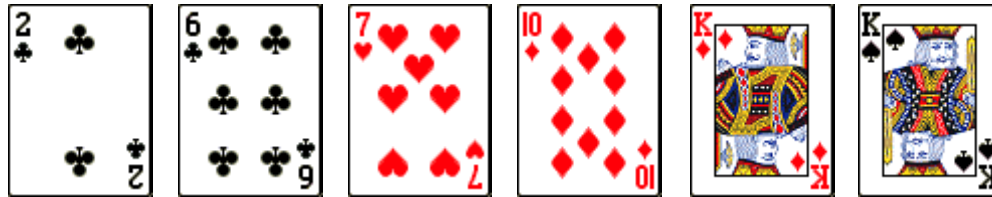
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted

Repeat



Sorted



Smallest





# Obvious Way

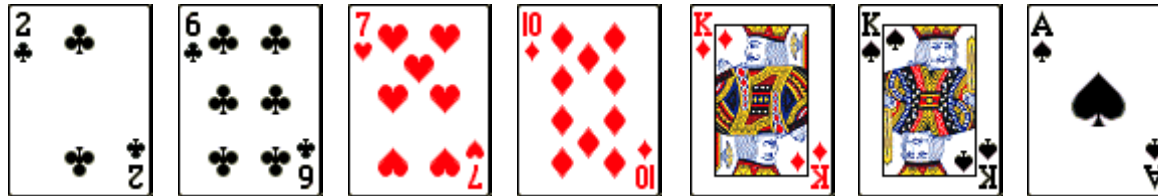
Find the smallest card not in hand (SCNIH), and put it at the end of your hand. Repeat.

Unsorted

Done

Sorted

Smallest



There is actually a name for this:

Selection Sort!

# Let's Implement it!

```
a = [4,12,3,1,11]
```

```
sort = []
```

```
while a:    # a is not []
    smallest = a[0]
    for element in a:
        if element < smallest:
            smallest = element
    a.remove(smallest)
    sort.append(smallest)
    print(a)
```

# Output

```
[4, 12, 3, 11]
```

```
[4, 12, 11]
```

```
[12, 11]
```

```
[12]
```

```
[]
```

```
print(a)
```

```
[]
```

```
print(sort)
```

```
[1, 3, 4, 11, 12]
```

# Order of Growth?

- Time:      Worst       $O(n^2)$   
                 Average       $O(n^2)$   
                 Best       $O(n^2)$
- Space:       ~~$O(n)$~~        $O(1)$

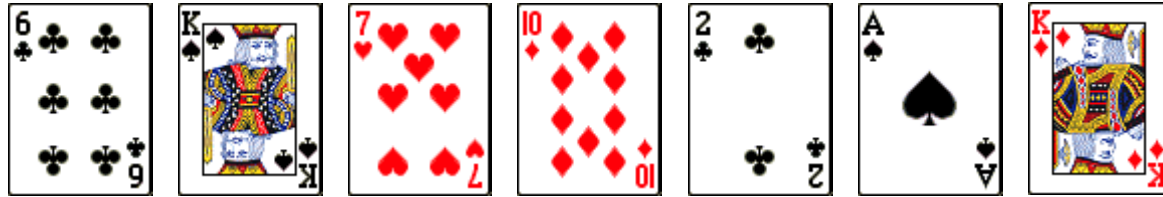
Let's try something  
else...

suppose you have a  
friend

# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

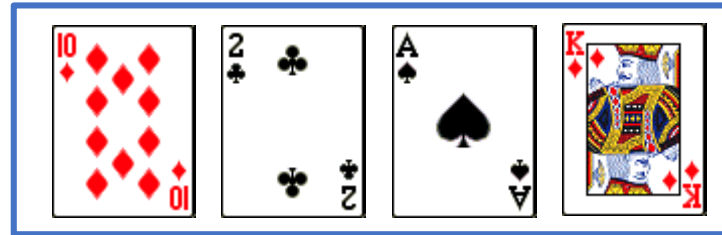
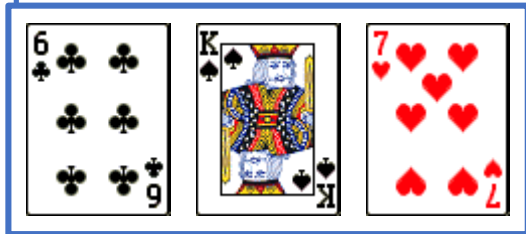
Split into halves



# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

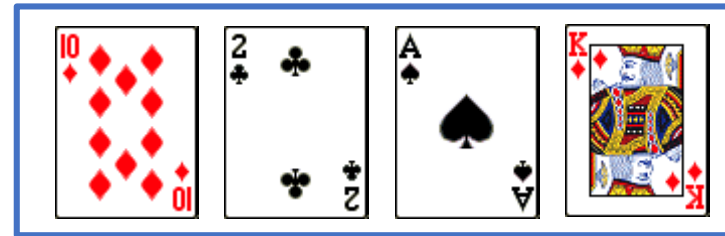
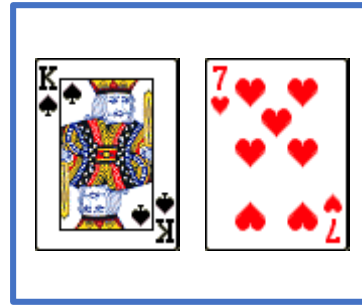
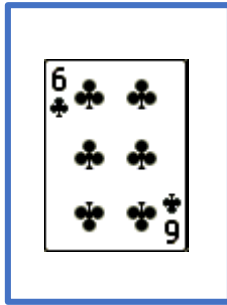
Split into halves





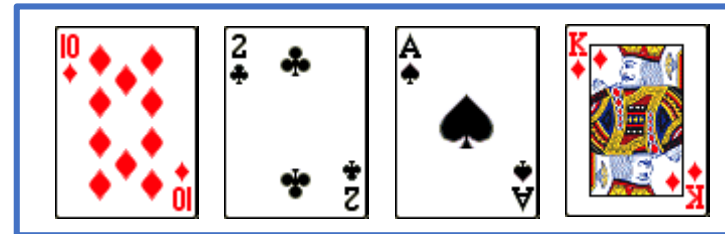
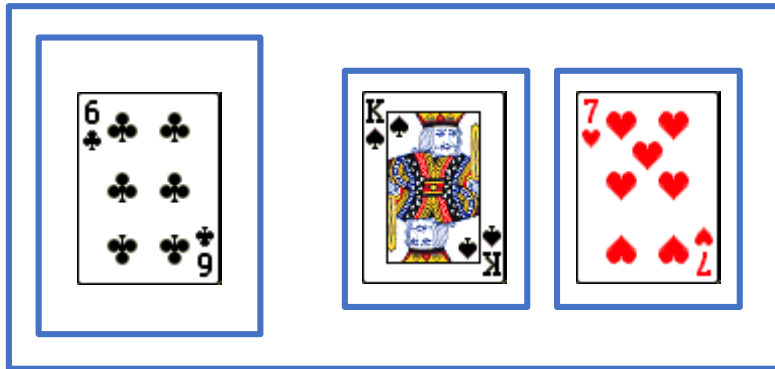
# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



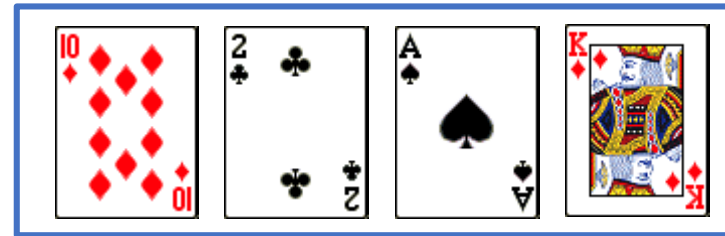
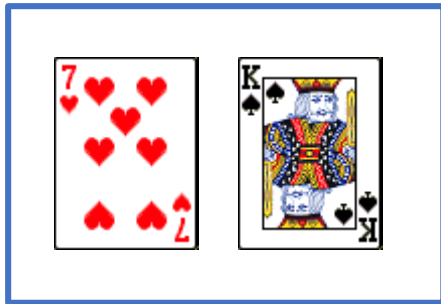
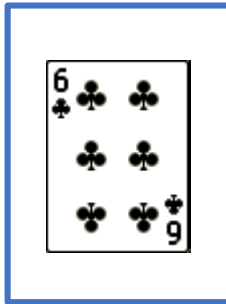
# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



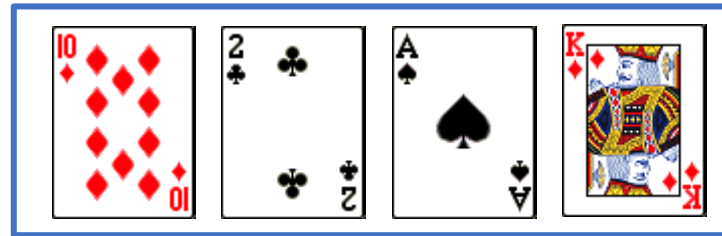
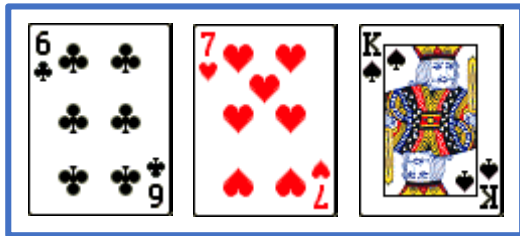
# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



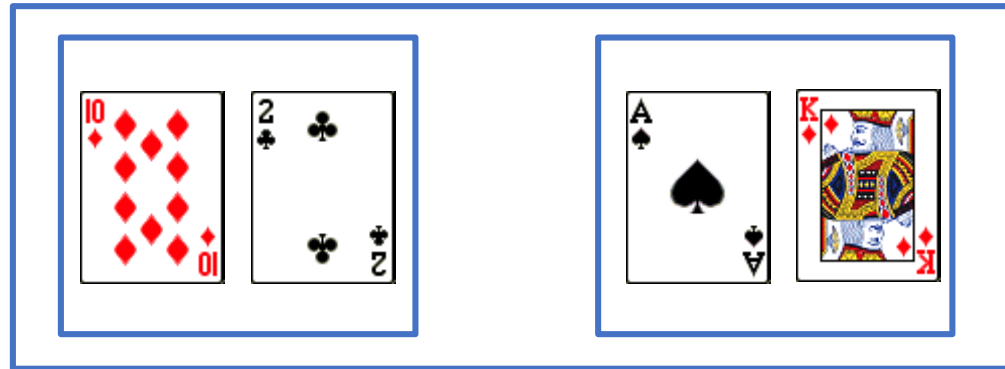
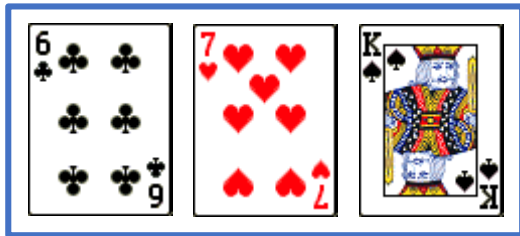
# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



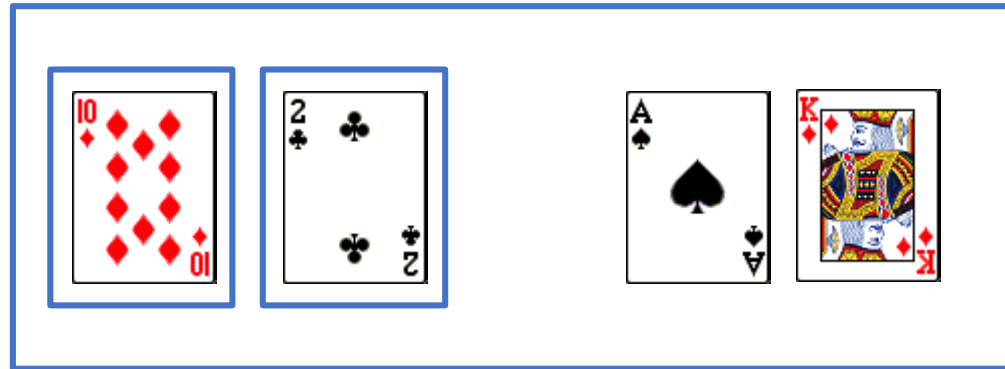
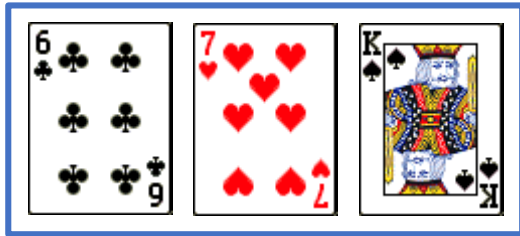
# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



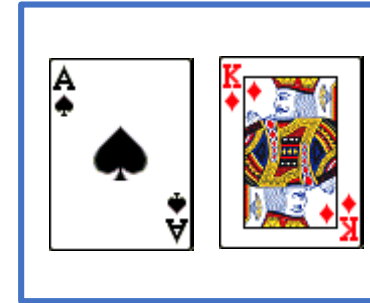
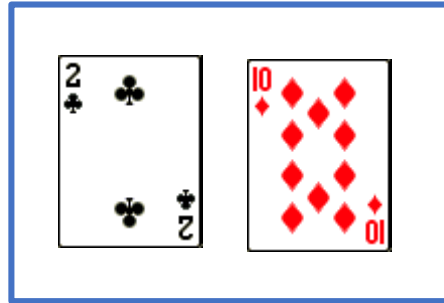
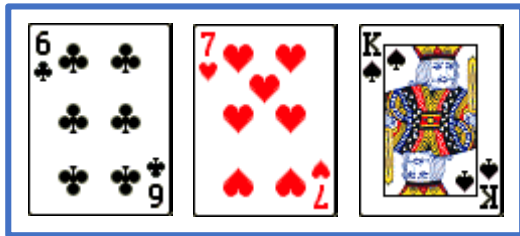
# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



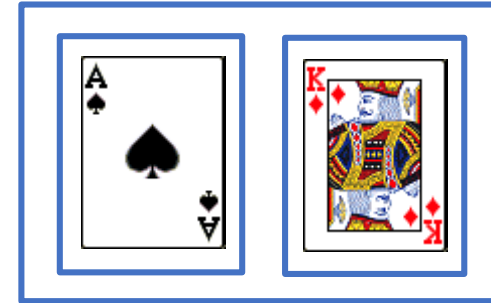
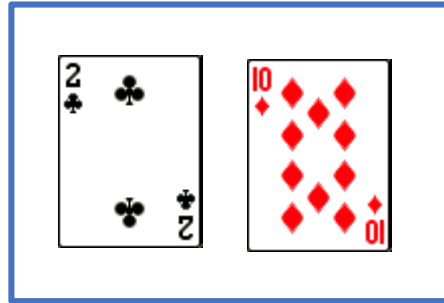
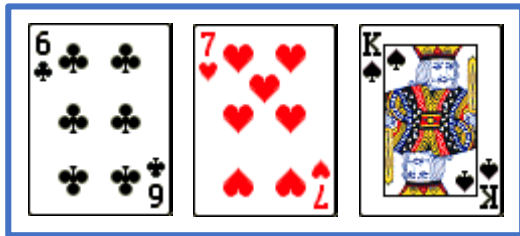
# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



# Doing it with a friend

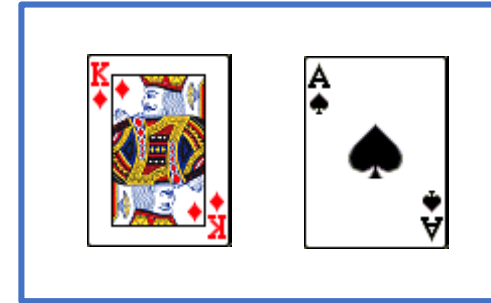
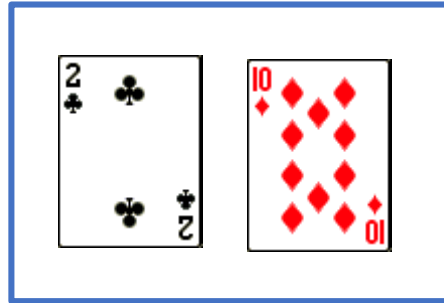
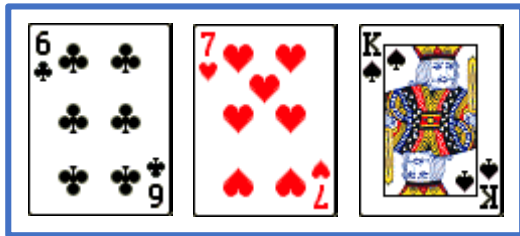
- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.





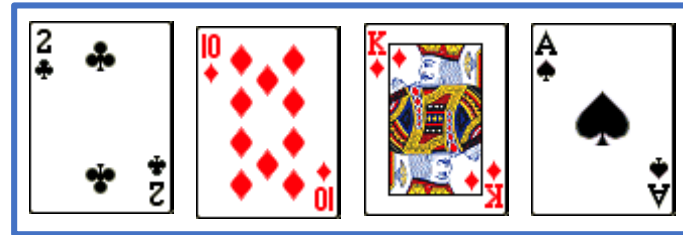
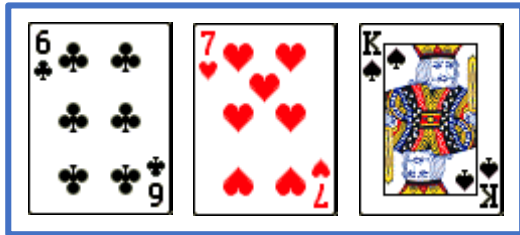
# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



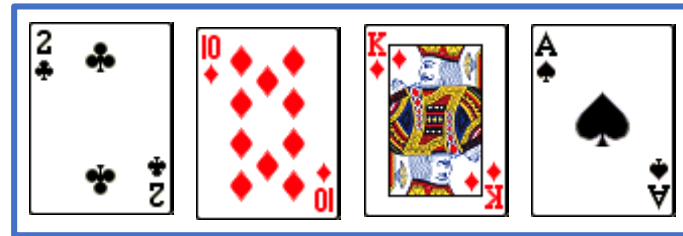
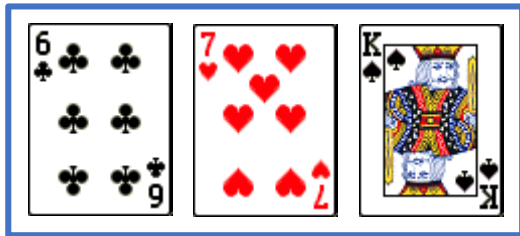
# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.



# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

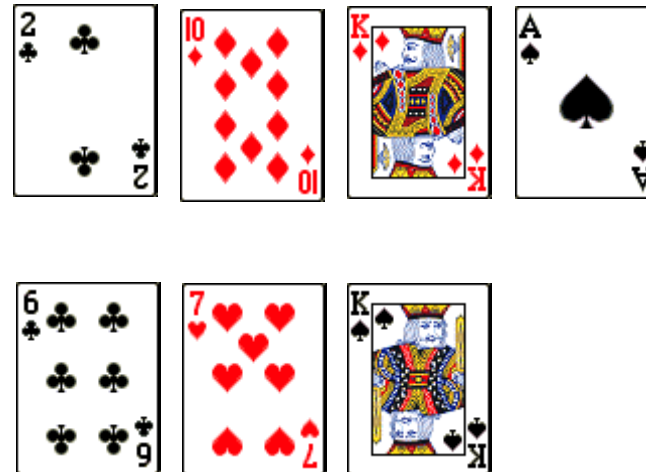


How to combine the 2 sorted halves?

# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

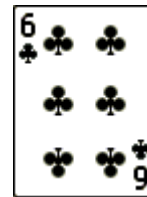
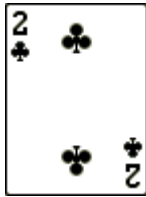
Compare first elements



# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

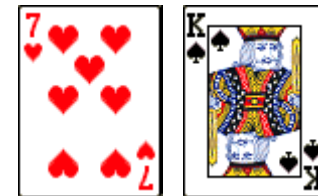
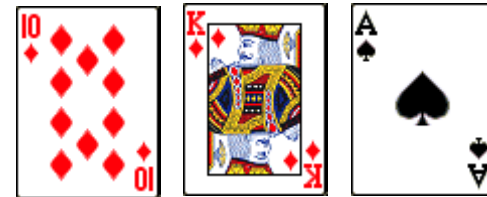
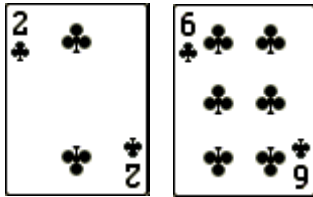
Compare first elements



# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

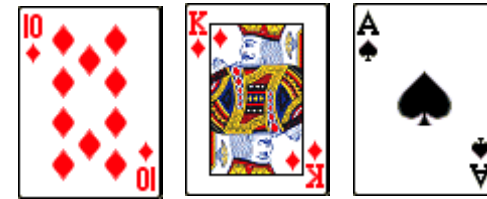
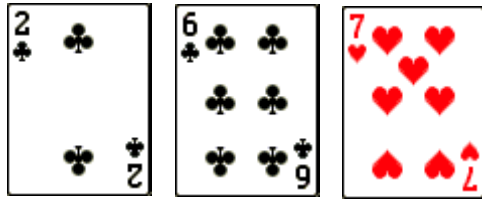
Compare first elements



# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

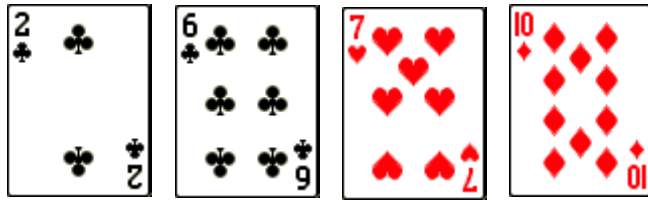
Compare first elements



# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Compare first elements

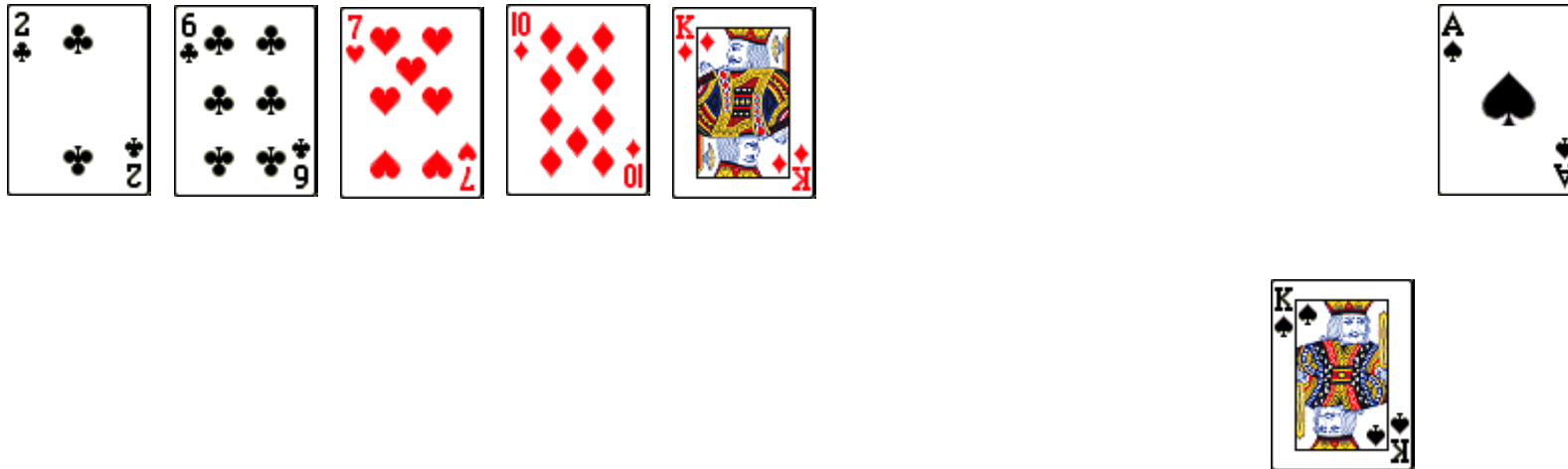




# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

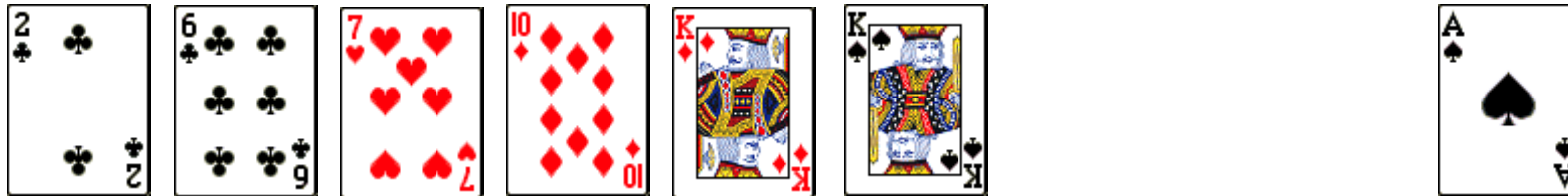
Compare first elements



# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

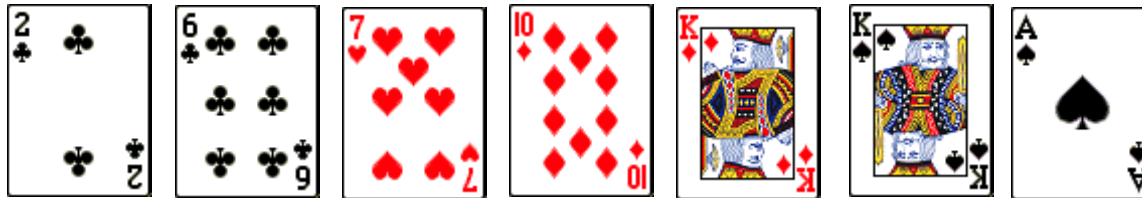
Compare first elements



# Doing it with a friend

- Split cards into two halves and sort. Combine halves afterwards. Repeat with each half.

Compare first elements



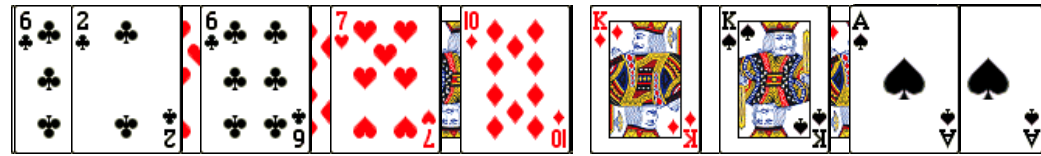
There is also a name for this:

Merge Sort!

# Let's Implement It!

First observation: **RECURSION!**

- Base case:  $n < 2$ , return list
- Otherwise:



- Divide list into two
- Sort each of them
- Merge!

# Merge Sort

```
def merge_sort(lst):  
    if len(lst) < 2:  # Base case!  
        return lst  
    mid = len(lst) // 2  
    left = merge_sort(lst[:mid])  #sort left  
    right = merge_sort(lst[mid:]) #sort right  
    return merge(left, right)
```

## How to merge?

# How to merge?

- Compare first element
- Take the smaller of the two
- Repeat until no more elements

# Merging

```
def merge(left, right):  
    results = []  
    while left and right:  
        if left[0] < right[0]:  
            results.append(left.pop(0))  
        else:  
            results.append(right.pop(0))  
    results.extend(left)  
    results.extend(right)  
    return results
```



# Order of Growth?

- Time:            Worst       $O(n \log n)$   
                      Average     $O(n \log n)$   
                      Best         $O(n \log n)$
- Space:             $O(n)$

No need to memorize

# Sort Properties

**In-place:** uses a small, constant amount of extra storage space, i.e.,  $O(1)$  space

**Selection Sort:** No (Possible)

**Merge Sort:** No (Possible)

# Sort Properties

**Stability:** maintains the relative order of items with equal keys (i.e., values)

Selection Sort: Yes (maybe)

Merge Sort: Yes

How Many Ways to  
Sort?

Too many. 😊

# How Many Sort Must You Learn?

None (sort of) 😊

```
list.sort()
```

No need to remember the  
time/space complexity

# list.sort

- `sort(*, key=None, reverse=None)`
  - This method sorts the list in place, using only < comparisons
  - Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).
  - `key` specifies a function of one argument that is used to extract a comparison key from each list element
  - `reverse` is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.
  - This method modifies (mutates) the sequence in place.
  - The `sort()` method is guaranteed to be stable

# Example

```
>>> a = [4, 32, 3, 34, 7, 31, 2, 1]
```

```
>>> a.sort()
```

```
>>> print(a)
```

```
[1, 2, 3, 4, 7, 31, 32, 34]
```

```
>>> a.sort(key=lambda x: x%5)
```

```
>>> print(a)
```

```
[1, 31, 2, 7, 32, 3, 4, 34]
```



# Sorting Records

Mostly not entirely too useful to sort values.  
Typically, we sort records using a key.

# Example

```
students = [  
    ('john', 'A', 15),  
    ('jane', 'B', 10),  
    ('ben', 'C', 8),  
    ('simon', 'A', 21),  
    ('dave', 'B', 12)]
```

# Example

```
students.sort()  
print(students)  
[('ben', 'C', 8),  
 ('dave', 'B', 12),  
 ('jane', 'B', 10),  
 ('john', 'A', 15),  
 ('simon', 'A', 21)]
```

# Example

```
students.sort(key=lambda x: x[2], reverse=True)
print(students)
[('simon', 'A', 21),
 ('john', 'A', 15),
 ('dave', 'B', 12),
 ('jane', 'B', 10),
 ('ben', 'C', 8)]
```

# Example

```
students.sort(key=lambda x: x[1])  
print(students)  
[('simon', 'A', 21),  
 ('john', 'A', 15),  
 ('dave', 'B', 12),  
 ('jane', 'B', 10),  
 ('ben', 'C', 8)]
```

# Example

```
students = [  
    ('john', 'A', 15),  
    ('jane', 'B', 10),  
    ('ben', 'C', 8),  
    ('simon', 'A', 21),  
    ('dave', 'B', 12)]
```

# Example

```
students.sort(key=lambda x: x[1])  
print(students)  
[('john', 'A', 15),  
 ('simon', 'A', 21),  
 ('jane', 'B', 10),  
 ('dave', 'B', 12),  
 ('ben', 'C', 8)]
```

Scipy

Numpy



# Summary

- Python Lists are mutable data structures
- Searching
  - Linear Search
  - Binary Search: Divide-and-conquer
- Sorting
  - Selection Sort
  - Merge Sort: Divide-and-conquer + recursion
  - Properties: In-place & Stability